Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 171–190

www.elsevier.com/locate/entcs

Precision and the Conjunction Rule in Concurrent Separation Logic

Alexey Gotsman[a] Josh Berdine[b] Byron Cook[b,c]

[a] *IMDEA Software Institute*

[b] *Microsoft Research*

[c] *Queen Mary University of London*

Abstract

Concurrent separation logic is a Hoare logic for modular reasoning about concurrent heap-manipulating programs synchronising via locks. It achieves modular reasoning by partitioning the program state into thread-local and lock-protected parts, and assigning resource invariants to the latter. Surprisingly, the logic is unsound unless resource invariants are precise, i.e., unambiguously carve out an area of the heap. The counterexample showing the unsoundness involves the conjunction rule. However, to date it has been an open question whether concurrent separation logic without the conjunction rule is sound when the restriction on resource invariants is dropped: all the published proofs have the precision restriction baked in. In this paper we present a single proof that shows the soundness of the logic with imprecise resource invariants, but without the conjunction rule, as well as its classical version, where resource invariants are required to be precise and the conjunction rule is included. Our proof yields a precise and direct formulation of O'Hearn's Separation Property and provides a semantic analysis of the logic that is much more elementary than previous proofs.

Keywords: Separation logic, concurrency, precision, conjunction rule.

1 Introduction

Concurrent separation logic [12] is a Hoare logic for modular reasoning about concurrent heap-manipulating programs synchronising via locks, aka mutexes. It achieves modular reasoning by imposing a partitioning of the variables and the heap forming the program state into several disjoint parts: thread-local parts (one for each thread, aka process) and protected parts (one for each free lock, i.e., a lock that is not held by any thread). A thread-local part may only be accessed by the corresponding thread, and a lock-protected part only when a thread holds the lock. When such a partitioning exists, the program is said to satisfy the *Separation Property*. To specify the partitioning, the logic associates each lock in the program with an assertion—its *resource invariant*—that describes the part of the state it protects. For example, a resource invariant might state that a lock protects a singly-linked list with the head node pointed to by a particular variable. For any given thread,

doi:10.1016/j.entcs.2011.09.021

resource invariants restrict how the other threads can change the protected state, and hence, allow reasoning about the thread in isolation.

It is important to note that the state partitioning described above is not a part of the program itself, but is enforced by proofs in the logic to enable modular reasoning in the presence of concurrent interference. Moreover, the partitioning is not required to be static, i.e., the logic permits ownership transfer of variables and heap cells between areas owned by different threads and locks. Such a non-standard view of the program state makes the formulation and proof of soundness of the logic difficult. In fact, the logic was proposed by O'Hearn in 2001, but the first proof of soundness (due to Brookes) was given only in 2004 [2]. This is because, as Reynolds showed shortly after the logic was invented [12], it is unsound unless resource invariants are *precise*. Informally, a predicate over program states is precise when it unambiguously carves out an area of the heap (see Section 2.1 for a formal definition). For example, the separation logic assertion $\mathtt{x} \mapsto 0$, denoting a cell at the address $\mathtt{x}$ storing 0, is precise; however, the assertion $\mathtt{x} \mapsto 0 \vee \mathsf{emp}$, denoting either the cell or the empty heap, is not. The key proof rule used in the counterexample showing the unsoundness is the *conjunction rule*:

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\} \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

The rule is useful for combining the results of two proofs; e.g., it is used by the reduced product construction in abstract interpretation [6].

O'Hearn has conjectured that the logic might be sound in the case when both the restriction of precision and the conjunction rule are dropped [12]. The question of this conjecture's validity is not only of theoretical importance: we do want to use imprecise resource invariants in practice. For example, consider the following two definitions of a list-segment predicate [15] we could use in a resource invariant:

$$\mathsf{ls}_1(E,F) \Leftrightarrow (E = F \wedge \mathsf{emp}) \vee (\exists X.\, E{\mapsto}X * \mathsf{ls}_1(X,F) \wedge E \neq F);$$
$$\mathsf{ls}_2(E,F) \Leftrightarrow (E = F \wedge \mathsf{emp}) \vee (\exists X.\, E{\mapsto}X * \mathsf{ls}_2(X,F)),$$

where X is chosen fresh. The latter definition yields imprecise assertions: e.g., both the heaps described by $X{\mapsto}Y$ and $\exists Z.\, X{\mapsto}Y * Y{\mapsto}Z * Z{\mapsto}Y$ satisfy $\mathsf{ls}_2(X,Y)$; only the former assertion implies $\mathsf{ls}_1(X,Y)$. However, the ls_2 predicate validates some entailments between assertions that ls_1 does not. For this reason, it is the ls_2 predicate that modern program analysis tools based on separation logic use (e.g., [16]). As automatic tools based on concurrent separation logic are usually built on top of corresponding sequential analyses, these tools thus often infer imprecise resource invariants [3,9].

To date, it has been an open question whether O'Hearn's conjecture is true: the whole of Brookes's proof of soundness [2] and all alternative proofs published after it [4,11] depend on the precision of resource invariants and thus have the conjunction rule baked in. The automatic program analyses producing imprecise invariants

have either been proved sound directly with respect to program semantics [9], or relied on the conjecture being true [3]. In this paper we present a single proof that shows the soundness of the logic with imprecise resource invariants, but without the conjunction rule, as well as its classical version, where resource invariants are required to be precise and the conjunction rule is included (Theorem 4.1). In addition to showing the soundness of concurrent separation logic, our proof provides a semantic analysis that is much more elementary than the previously proposed ones, which have been based on action traces [2,4] or Petri nets [11].

We achieve this using the concept of a *semantic proof* that annotates program points in the code of a thread with descriptions of its local state (Section 4.1). Unlike the standard interpretations of Hoare triples, the interpretation in terms of semantic proofs is quite intentional. The definition of a triple being valid does not abstract away all the internal syntactic structure of the command or proof, while the standard interpretations are given solely in terms of the extensional meaning of the command and pre- and post-condition assertions. This use of an intentional definition is an acknowledgement that the intuitive reason for the unsoundness of the conjunction rule with imprecise resource invariants is crucially about *proofs*, not denotations of commands. In particular, imprecise resource invariants allow the two premisses of the conjunction rule to make conflicting choices about how to partition the state. It is these different choices of state partitioning in different branches of the proof that lead to problems, but the partitioning in question is irrelevant to the operational behaviour of the command.

One view of previous soundness proofs is that in lieu of using an intentional interpretation of triples, they instrument the semantics of commands with manipulation of the partitioning in order to expose enough of the intentional detail even with an extensional interpretation. A key consequence of using semantic proofs instead of an instrumented semantics is that the Separation Property can be formulated (Lemma 4.2) directly as an invariant of concurrent executions: for a given point in the program the local states of threads can be determined from their semantic proofs. Consequently, no tracking of changing instrumentation along execution traces is needed, and the previously crucial and difficult step of decomposing a concurrent execution into an interleaving of constituent sequential executions (Brookes's Parallel Decomposition Lemma [2]) can be avoided.

Technically, to prove the soundness of concurrent separation logic, we define a *thread-local interpretation* of every thread in the program as a semantic proof. A formalisation of the Separation Property (Lemma 4.2) connects the thread-local interpretation to a standard interleaving operational semantics (Section 3). We then define the notion of validity of Hoare triples for commands with respect to this interpretation and prove the soundness of all the proof rules (Section 4.2). Despite not tracking a partitioning of the state, the thread-local interpretation is strong enough to establish that provability of a program in concurrent separation logic implies that the program is data-race free (Section 5).

2 Concurrent separation logic

In this paper we consider the version of concurrent separation logic proposed by Calcagno et al. [4]. This version of the logic is *abstract* in the sense that it can be interpreted over a wide class of semantic models with a given structure, which allows reusing results about the logic in multiple contexts. As any Hoare logic, concurrent separation logic includes two formal systems—one for assertions and one for specifications. We discuss the former first.

2.1 Assertions

In abstract separation logic, assertions are interpreted with respect to a separation algebra, which represents program states.

Definition 2.1 (Separation algebra) *A separation algebra is a partial commutative monoid $(\Sigma, *, \varepsilon)$ with a unit element $\varepsilon \in \Sigma$. A partial commutative monoid is given by a partial binary operation of separate combination $*$, where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined.*

The original definition of separation algebras given in [4] requires the $*$ operation to be *cancellative*: for each $\sigma \in \Sigma$, the partial function $\sigma * \cdot : \Sigma \rightharpoonup \Sigma$ must be injective. This requirement is connected with conditions for validating the conjunction rule of Hoare logic. We have omitted it here since we also consider models of concurrent separation logic invalidating the rule.

In this paper, by a *domain* D we understand a lattice $(D, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$. For a set Σ let $\mathcal{P}(\Sigma)^\top$ be the domain of subsets of Σ with a special element $\top$. The order $\sqsubseteq$ in the domain $\mathcal{P}(\Sigma)^\top$ is subset inclusion with $\top$ being the greatest element and $\emptyset$ the least. When Σ represents program states, we usually use $\top$ to denote an error state, e.g., resulting from dereferencing an invalid pointer. Note that the order $\sqsubseteq$ defines the corresponding join $\sqcup$ and meet $\sqcap$ operations on the domain $\mathcal{P}(\Sigma)^\top$. If Σ is a separation algebra, we can lift the $*$ operation to $\mathcal{P}(\Sigma)^\top$ pointwise: for all $p, q \in \mathcal{P}(\Sigma)$

$$p * q = \bigcup \{\sigma * \eta \mid \sigma \in p, \eta \in q, \sigma * \eta \text{ is defined}\}; \quad \top * p = p * \top = \top.$$

Thus, $\mathcal{P}(\Sigma)^\top$ has a total commutative monoid structure with the unit $e = \{\varepsilon\}$. For a separation algebra Σ, we call $\mathcal{P}(\Sigma)^\top$ the *separation domain* constructed from the algebra Σ.

We denote with $\circledast$ the iterated version of $*$ on $\mathcal{P}(\Sigma)^\top$: $\overset{n}{\underset{k=1}{\circledast}} p_k = e * p_1 * \ldots * p_n$. For $\sigma \in \Sigma \cup \{\top\}$ we denote with $\{\!|\sigma|\!\}$ the singleton set $\{\sigma\}$, if $\sigma \in \Sigma$, and $\top$, if $\sigma = \top$. Thus, $\{\!|\sigma|\!\} \in \mathcal{P}(\Sigma)^\top$.

A predicate $p \in \mathcal{P}(\Sigma)$ over a separation algebra Σ is *precise* [12,13] if for any state σ there exists at most one substate σ_1 satisfying p: $\sigma = \sigma_1 * \sigma_2$ for some σ_2. If such a substate exists and the $*$ operation is cancellative, then the substate σ_2 is unique.

Elements of separation algebras and domains are often defined using partial functions. We use the following notation: $f(x)\uparrow$ means that the function f is undefined on x, and $[\,]$ denotes a nowhere-defined function. We denote with $f[x : y]$ the function that has the same value as f everywhere, except for x, where it has the value y (even if $f(x)\uparrow$).

The following is an example of a separation algebra RAM typically used for reasoning about heap-manipulating programs:

$$\begin{array}{lll} \text{Values} = \mathbb{Z} & \text{Vars} = \{\mathtt{x}, \mathtt{y}, \ldots\} & \text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \text{Values} \\ \text{Locs} = \mathbb{N} & \text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} \text{Values} & \mathsf{RAM} = \text{Stacks} \times \text{Heaps} \end{array}$$

A state consists of a stack and a heap, both finite partial functions mapping variables or locations to their values. To simplify presentation, the algebra does not include permissions [1,14]. We have also omitted logical variables; see Section 2.4. The $*$ operation forms the disjoint union of stacks and heaps: $(s_1, h_1) * (s_2, h_2) = (s_1 \uplus s_2, h_1 \uplus h_2)$. The unit element is a state with the empty stack and heap: $([\,], [\,])$.

For the remainder, we fix a separation algebra $(\Sigma, *, \varepsilon)$ and the corresponding domain $\mathcal{P}(\Sigma)^\top$. We further assume an assertion language for denoting predicates over Σ, including $\vee$, $\wedge$, $\Rightarrow$ and $*$ connectives with the expected interpretation, and the assertion emp denoting only the empty state ε. Tautological assertions are those whose meaning is Σ. We denote with $[\![P]\!] \in \mathcal{P}(\Sigma)$ the meaning of the assertion P. An assertion is precise if it denotes a precise predicate.

2.2 Primitive commands and local functions

The programming language we consider in this paper is parameterised by a set PComm of primitive sequential commands. For every $C \in \mathsf{PComm}$ we assume its denotation $f_C : \Sigma \to \mathcal{P}(\Sigma)^\top$, which maps each pre-state to the states obtained by executing C from it. As shown by Calcagno et al. [4], for separation logic to be sound, transformers f_C for primitive commands of the programming language must behave in a local way with respect to the structure present in Σ. The following definition formalises this condition.

Definition 2.2 (Local function) *For a separation algebra $(\Sigma, *, \varepsilon)$, a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is local if for any states $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1 * \sigma_2$ is defined, we have*

$$f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}.$$

Definition 2.2 is a concise way of formulating two conditions that the soundness of separation logic relies on [17]: if $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is the meaning of a command C, then

(safety monotonicity) if executing C from a state $\sigma_1 * \sigma_2$ results in an error $f(\sigma_1 * \sigma_2) = \top$, then executing C from a smaller state σ_1 also produces an error: $\top \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ implies $f(\sigma_1) = \top$;

(frame property) if executing C from a state σ_1 does not produce an error, then executing C from a larger state $\sigma_1 * \sigma_2$, has the same effect and leaves σ_2 unchanged: in this case we often have $f(\sigma_1 * \sigma_2) = f(\sigma_1) * \{\sigma_2\}$.

The requirement of locality rules out commands that can check if a cell is allocated in the heap other than by trying to access it and faulting if it is not allocated. For example, let $\Sigma = \mathsf{RAM}$ (see Section 2.1) and consider the following function $f : \mathsf{RAM} \to \mathcal{P}(\mathsf{RAM})^\top$:

$$f(s,h) = \begin{cases} \{(s,h')\}, & \text{if } h(10) \text{ is defined;} \\ \{(s,h)\}, & \text{otherwise,} \end{cases}$$

where h' is identical to h except it is undefined at 10. The function f defines the denotation of a 'command' that disposes of the cell at the address 10 if it is allocated and acts as a no-op if it is not. The function f is not local: take $\sigma_1 = ([\,],[\,])$ and $\sigma_2 = ([\,],[10:0])$, then

$$f(\sigma_1 * \sigma_2) = f(([\,],[\,]) * ([\,],[10:0])) = f([\,],[10:0]) = \{([\,],[\,])\}$$

and

$$f(\sigma_1) * \{\sigma_2\} = f([\,],[\,]) * \{([\,],[10:0])\} = \\ \{([\,],[\,])\} * \{([\,],[10:0])\} = \{([\,],[10:0])\},$$

hence, the inequality $f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ does not hold.

The *pointwise lifting* of a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ to $\mathcal{P}(\Sigma)^\top$ is a function $f : \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$ defined as follows: for all $p \in \mathcal{P}(\Sigma)^\top$

$$f(p) = \begin{cases} \bigsqcup\{f(\sigma) \mid \sigma \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top. \end{cases}$$

Given a denotation $f_C : \Sigma \to \mathcal{P}(\Sigma)^\top$ of a command $C \in \mathsf{PComm}$, we can lift it to a forward predicate transformer $f_C : \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$. We note that the resulting transformer distributes over the $\sqcup$ and $\sqcap$ operations in the domain $\mathcal{P}(\Sigma)^\top$:

$$\forall p, q \in \mathcal{P}(\Sigma)^\top . f_C(p \sqcup q) = f_C(p) \sqcup f_C(q); \tag{1}$$
$$\forall p, q \in \mathcal{P}(\Sigma)^\top . f_C(p \sqcap q) = f_C(p) \sqcap f_C(q). \tag{2}$$

Furthermore, if the denotation is local, then for the corresponding transformer we have:

$$\forall p, q \in \mathcal{P}(\Sigma)^\top . f_C(p * q) \sqsubseteq f_C(p) * q.$$

We say that the predicate transformer is local when it satisfies this property.

Typical heap-manipulating commands can be interpreted over the algebra $\Sigma = \mathsf{RAM}$ from Section 2.1. We refer to them in Section 5, where we formulate and prove

$$
\begin{array}{lcll}
\texttt{skip}, (s,h) & \rightsquigarrow & (s,h) & \\
\texttt{x=}E, (s[\texttt{x}:u],h) & \rightsquigarrow & (s[\texttt{x}:[\![E]\!]s[\texttt{x}:u]],h) & \\
\texttt{x=}[E], (s[\texttt{x}:u],h[[\![E]\!]s[\texttt{x}:u]:b]) & \rightsquigarrow & (s[\texttt{x}:b],h[[\![E]\!]s[\texttt{x}:u]:b]), & \\
[E]\texttt{=}F, (s,h[[\![E]\!]s:u]) & \rightsquigarrow & (s,h[[\![E]\!]s:[\![F]\!]s]) & \\
\texttt{x=new}, (s[\texttt{x}:u],h) & \rightsquigarrow & (s[\texttt{x}:b],h[b:w]), \text{ if } h(b)\uparrow & \\
\texttt{delete } E, (s,h[[\![E]\!]s:u]) & \rightsquigarrow & (s,h), \text{ if } h([\![E]\!]s)\uparrow & \\
\texttt{assume}(B), (s,h) & \rightsquigarrow & (s,h), \text{ if } [\![B]\!]s = \textbf{true} & \\
\texttt{assume}(B), (s,h) & \not\rightsquigarrow & \text{if } [\![B]\!]s = \textbf{false} & \\
C, (s,h) & \rightsquigarrow & \top, \quad \text{otherwise} &
\end{array}
$$

Fig. 1. Transition relation for primitive commands RAMComm. $\top$ indicates that the command faults. $\not\rightsquigarrow$ is used to denote that the command does not fault, but gets stuck. We denote with $[\![E]\!]s \in$ Values and $[\![B]\!]s \in \{\textbf{true}, \textbf{false}\}$ the values of the expressions in the stack s.

the data-race freedom theorem for concurrent separation logic. Let E, F range over integer expressions and B over Boolean expressions:

$$
\begin{array}{rcl}
\texttt{x} & \in & \text{Vars} \\
E, F & ::= & \texttt{NULL} \mid \texttt{x} \mid E + F \mid \ldots \\
B & ::= & E = F \mid \neg B \mid \ldots
\end{array}
$$

We consider the following set RAMComm of primitive sequential commands:

$$\text{RAMComm} ::= \texttt{skip} \mid \texttt{x=}E \mid \texttt{x=}[E] \mid [E]\texttt{=}F \mid \texttt{x=new} \mid \texttt{delete } E \mid \texttt{assume}(B)$$

As usual, square brackets denote pointer dereferencing. The $\texttt{assume}(B)$ command acts as a filter on the state space of programs—B is assumed to be true after $\texttt{assume}(B)$ is executed. We define denotations $f_C : \text{RAM} \to \mathcal{P}(\text{RAM})^\top$ for $C \in \text{RAMComm}$ using the transition relation $\rightsquigarrow$: RAMComm $\times$ RAM $\times$ (RAM $\cup \{\top\}$) shown in Figure 1: for all $\sigma \in \text{RAM}$

$$f_C(\sigma) = \bigsqcup \{ \{\!|\sigma'|\!\} \mid C, \sigma \rightsquigarrow \sigma' \}.$$

It is not difficult to show that f_C is local for every $C \in \text{RAMComm}$ [4].

In the rest of this paper, we assume local denotations $f_C : \Sigma \to \mathcal{P}(\Sigma)^\top$ of commands $C \in \text{PComm}$ and their liftings to predicate transformers.

2.3 The logic

We consider a variant of concurrent separation logic [12] for a concurrent programming language in which programs consist of a parallel composition of several threads

(processes) that use locks (mutexes) $\ell_1, \ldots, \ell_m$ for synchronisation. The syntax of programs S is as follows:

$$\begin{aligned} C &::= \mathsf{PComm} \mid C; C \mid C + C \mid C^* \mid \mathtt{acquire}(\ell_k);\, C;\, \mathtt{release}(\ell_k) \\ S &::= C \parallel \ldots \parallel C \end{aligned}$$

The code of threads can include primitive sequential commands from PComm, sequential composition $C; C$, choice $C + C$, iteration C^* and (syntactically scoped) critical regions over the available locks.

When PComm includes the `assume` statement, the standard commands for conditionals, loops and conditional critical regions (CCRs) can be defined in our programming language as follows:

$$\begin{aligned} \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 &= (\mathtt{assume}(B); C_1) + (\mathtt{assume}(\neg B); C_2) \\ \mathtt{while}\ B\ \mathtt{do}\ C &= (\mathtt{assume}(B); C)^*; \mathtt{assume}(\neg B) \\ \mathtt{with}\ \ell\ \mathtt{when}\ B\ \mathtt{do}\ C &= \mathtt{acquire}(\ell); \mathtt{assume}(B); C; \mathtt{release}(\ell) \end{aligned}$$

The original concurrent separation logic also considers nested parallel compositions and explicit lock declarations. The restricted form of programs chosen here simplifies the formal development and makes the underlying ideas more explicit. Our results have been extended to dynamically-allocated locks and dynamically-created threads (see [8]), which are more general constructs than lock declarations and parallel compositions.

The judgements of concurrent separation logic are of the form $I \vdash \{P\}\ C\ \{Q\}$, where C is a command in the code of a thread, P and Q are assertions describing the local state of the thread and I is the vector of resource invariants I_k for all the locks ℓ_k in the program. Intuitively, the judgement means that, if the thread starts executing C from an initial local state satisfying P, then it accesses only its local part of the state, respects the resource invariants I, and terminates only in local states satisfying Q.

The proof rules of concurrent separation logic are summarised in Figure 2. Most of the rules are standard ones from Hoare logic. We have a single axiom for primitive commands (Prim), which allows any pre- and postconditions consistent with the predicate transformer for the command. For a particular set of states Σ and denotations f_C of $C \in \mathsf{PComm}$, this axiom can be specialised to several syntactic versions, obtaining a concrete instance of the abstract logic presented here [1,14,15]. The conjunction rule (Conj) is useful for combining the results of two proofs, and the disjunction rule (Disj) for proof by cases. The frame rule (Frame) states that executing a command in a bigger local state does not change its behaviour.

Locks are treated in the logic as follows. When a thread acquires a lock, it receives the ownership of a part of the state satisfying the resource invariant of the lock (Acquire). Before releasing the lock, the thread must re-establish the corresponding resource invariant. After the lock is released, the thread relinquishes

$$\frac{f_C([\![P]\!]) \sqsubseteq [\![Q]\!]}{I \vdash \{P\}\ C\ \{Q\}}\ \textsc{Prim}$$

$$\frac{I \vdash \{P\}\ C_1\ \{Q\} \quad I \vdash \{Q\}\ C_2\ \{R\}}{I \vdash \{P\}\ C_1; C_2\ \{R\}}\ \textsc{Seq}$$

$$\frac{I \vdash \{P\}\ C_1\ \{Q\} \quad I \vdash \{P\}\ C_2\ \{Q\}}{I \vdash \{P\}\ C_1 + C_2\ \{Q\}}\ \textsc{Choice}$$

$$\frac{I \vdash \{P\}\ C\ \{P\}}{I \vdash \{P\}\ C^*\ \{P\}}\ \textsc{Loop}$$

$$\frac{I \vdash \{P_1\}\ C\ \{Q_1\} \quad I \vdash \{P_2\}\ C\ \{Q_2\}}{I \vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{I \vdash \{P_1\}\ C\ \{Q_1\} \quad I \vdash \{P_2\}\ C\ \{Q_2\}}{I \vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}\ \textsc{Conj}$$

$$\frac{P_1 \Rightarrow P_2 \quad I \vdash \{P_2\}\ C\ \{Q_2\} \quad Q_2 \Rightarrow Q_1}{I \vdash \{P_1\}\ C\ \{Q_1\}}\ \textsc{Conseq}$$

$$\frac{I \vdash \{P\}\ C\ \{Q\}}{I \vdash \{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame}$$

$$\frac{}{I \vdash \{\mathtt{emp}\}\ \mathtt{acquire}(\ell_k)\ \{I_k\}}\ \textsc{Acquire}$$

$$\frac{}{I \vdash \{I_k\}\ \mathtt{release}(\ell_k)\ \{\mathtt{emp}\}}\ \textsc{Release}$$

$$\frac{I \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \ldots \quad I \vdash \{P_n\}\ C_n\ \{Q_n\}}{I \vdash \{P_1 * \ldots * P_n\}\ C_1 \parallel \ldots \parallel C_n\ \{Q_1 * \ldots * Q_n\}}\ \textsc{Par}$$

Fig. 2. Proof rules of concurrent separation logic

the ownership of its resource invariant (Release). Note that we can obtain global versions of the axioms Acquire and Release by closing them under the frame rule:

$$\frac{}{I \vdash \{P\}\ \mathtt{acquire}(\ell_k)\ \{P * I_k\}} \qquad \frac{}{I \vdash \{P * I_k\}\ \mathtt{release}(\ell_k)\ \{P\}}$$

Finally, the Par rule combines judgements about several threads into a judgement for the whole program of the form $I \vdash \{P\}\ S\ \{Q\}$.

2.4 Logical variables

In program proofs we often need to use so-called logical (aka ghost) variables, which appear in assertions, but not in programs. We now show how the logic can be extended with proof rules for manipulating such variables.

Let us fix a set of integer logical variables LVars $= \{X, Y, \ldots\}$ and let Ints $=$

LVars $\to \mathbb{Z}$ be the set of their interpretations. We say that a separation algebra Σ is an *algebra with logical variables*, if for some separation algebra Σ' we have $\Sigma = \Sigma' \times \text{Ints}$ and the $*$ operation on Σ is defined as follows:

$$(\sigma_1, \mathbf{i}_1) * (\sigma_2, \mathbf{i}_2) = (\sigma_1 * \sigma_2, \mathbf{i}_1),$$

if $\mathbf{i}_1 = \mathbf{i}_2$, and is undefined, otherwise.

For example, let $\mathsf{RAM}' = \mathsf{RAM} \times \text{Ints}$ for the separation algebra RAM defined in Section 2.1 with the $*$ operation on RAM lifted to RAM' as above. Then RAM' is a separation algebra with logical variables.

Given a function $f : \Sigma' \to \mathcal{P}(\Sigma')^\top$ on the underlying algebra without logical variables, we can lift it to a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ on the algebra with logical variables as follows:

$$f(\sigma, \mathbf{i}) = \begin{cases} f(\sigma) \times \{\mathbf{i}\}, & \text{if } f(\sigma) \neq \top; \\ \top, & \text{if } f(\sigma) = \top. \end{cases}$$

Let Σ be an algebra with logical variables, and assume an assertion language with quantifiers over logical variables:

$$P \ ::= \ \ldots \mid \exists X.\, P \mid \forall X.\, P$$

where the satisfaction relation is defined as follows:

$$(\sigma, \mathbf{i}) \models \exists X.\, P \Leftrightarrow \exists u.\, (\sigma, \mathbf{i}[X : u]) \models P$$
$$(\sigma, \mathbf{i}) \models \forall X.\, P \Leftrightarrow \forall u.\, (\sigma, \mathbf{i}[X : u]) \models P$$

When the functions f_C defining the semantics of primitive sequential commands are lifted from functions on the underlying algebra without logical variables, we can extend concurrent separation logic with the following two proof rules for manipulating logical variables:

$$\frac{I \vdash \{P\}\ C\ \{Q\}}{I \vdash \{\exists X.\, P\}\ C\ \{\exists X.\, Q\}}\ \textsc{Exists} \qquad \frac{I \vdash \{P\}\ C\ \{Q\}}{I \vdash \{\forall X.\, P\}\ C\ \{\forall X.\, Q\}}\ \textsc{Forall}$$

3 Programming language and semantics

We now define the simple operational semantics with respect to which we prove soundness. From now on, we fix a program $S = C_1 \parallel \ldots \parallel C_n$ in our concurrent programming language consisting of n threads $C_1, \ldots, C_n$ that use m locks $\ell_1, \ldots, \ell_m$ for synchronisation.

It is technically convenient to abstract from the particular syntax of the programming language and represent each thread in a program with its control-flow graph (CFG). A CFG is defined as a tuple $(N, T, \mathsf{start}, \mathsf{end})$, where N is the set of program points, $T \subseteq N \times \mathsf{Comm} \times N$ is the control-flow relation, and start and end

are distinguished initial and final program points. Edges in the CFG are labelled with commands from the set Comm, which consists of primitive sequential commands PComm, lock acquires $\texttt{acquire}(\ell_k)$ and releases $\texttt{release}(\ell_k)$. We assume, without loss of generality, that control-flow relations have no edges leading to start or from end.

We note that the code of a thread in our language can be translated to a CFG in a standard way. Namely, assume the set PComm of primitive sequential commands includes the `skip` statement. Then the CFG of a command C is constructed by induction on its syntax as follows:

(i) A primitive command $C \in \mathsf{PComm}$ has the CFG

$$(\{\mathsf{start}, \mathsf{end}\}, \{(\mathsf{start}, C, \mathsf{end})\}, \mathsf{start}, \mathsf{end}).$$

(ii) Assume C_1 and C_2 have CFGs

$$(N_1, T_1, \mathsf{start}_1, \mathsf{end}_1) \quad \text{and} \quad (N_2, T_2, \mathsf{start}_2, \mathsf{end}_2), \tag{3}$$

respectively. Then $C_1; C_2$ has the CFG

$$(N_1 \cup N_2, T_1 \cup T_2 \cup \{(\mathsf{end}_1, \texttt{skip}, \mathsf{start}_2)\}, \mathsf{start}_1, \mathsf{end}_2).$$

(iii) Assume C_1 and C_2 have CFGs (3). Then $C_1 + C_2$ has the CFG

$$(N_1 \cup N_2 \cup \{\mathsf{start}, \mathsf{end}\}, T_1 \cup T_2 \cup \{(\mathsf{start}, \texttt{skip}, \mathsf{start}_1), (\mathsf{start}, \texttt{skip}, \mathsf{start}_2),$$
$$(\mathsf{end}_1, \texttt{skip}, \mathsf{end}), (\mathsf{end}_2, \texttt{skip}, \mathsf{end})\}, \mathsf{start}, \mathsf{end}).$$

(iv) Assume C has a CFG $(N, T, \mathsf{start}, \mathsf{end})$. Then C^* has the CFG

$$(N \cup \{\mathsf{start}', \mathsf{end}'\}, T \cup \{(\mathsf{start}', \texttt{skip}, \mathsf{start}), (\mathsf{end}, \texttt{skip}, \mathsf{start}),$$
$$(\mathsf{end}, \texttt{skip}, \mathsf{end}')\}, \mathsf{start}', \mathsf{end}').$$

Thus, let us represent each thread C_k in S by its CFG $(N_k, T_k, \mathsf{start}_k, \mathsf{end}_k)$. Let $N = \biguplus_{k=1}^{n} N_k$ and $T = \biguplus_{k=1}^{n} T_k$ be the set of program points and the control-flow relation of the program S, respectively.

The interleaving operational semantics of the program S is defined by a transition relation $\rightarrow_S$ that transforms pairs of program counters (represented by mappings from thread identifiers to program points) and program states:

$$\rightarrow_S : ((\{1, \ldots, n\} \to N) \times \Sigma) \times ((\{1, \ldots, n\} \to N) \times (\Sigma \cup \{\top\})).$$

Note that since the critical regions formed by `acquire` and `release` commands are syntactically scoped in our programming language, we can determine the set $\mathsf{Free}(\mathsf{pc}) \subseteq \{1, \ldots, m\}$ of indices of free locks at every program counter $\mathsf{pc} \in \{1, \ldots, n\} \to N$, i.e., the set of locks that are not held by any thread. The relation $\rightarrow_S$ is defined by the rules in Figure 3. The semantics executes commands

$$\frac{(v, C, v') \in T \quad C \in \mathsf{PComm} \quad f_C(\sigma) \neq \top \quad \sigma' \in f_C(\sigma)}{\mathsf{pc}[k : v], \sigma \to_S \mathsf{pc}[k : v'], \sigma'}$$

$$\frac{(v, \mathtt{release}(\ell_j), v') \in T}{\mathsf{pc}[k : v], \sigma \to_S \mathsf{pc}[k : v'], \sigma}$$

$$\frac{(v, C, v') \in T \quad C \in \mathsf{PComm} \quad f_C(\sigma) = \top}{\mathsf{pc}[k : v], \sigma \to_S \mathsf{pc}[k : v'], \top}$$

$$\frac{(v, \mathtt{acquire}(\ell_j), v') \in T \quad j \in \mathsf{Free}(\mathsf{pc}[k : v])}{\mathsf{pc}[k : v], \sigma \to_S \mathsf{pc}[k : v'], \sigma}$$

Fig. 3. Operational semantics

from PComm atomically. Note also that, according to our semantics, a thread that tries to acquire the same lock twice gets stuck.

Let us denote with pc_0 the initial program counter $[1 : \mathsf{start}_1] \dots [n : \mathsf{start}_n]$ and with $\mathsf{pc_f}$ the final one $[1 : \mathsf{end}_1] \dots [n : \mathsf{end}_n]$. We say that the program S is *safe* when run from an initial state $\sigma_0 \in \Sigma$, if it is not the case that $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \top$ for some program counter pc.

4 Proof of soundness

The purpose of this section is to prove the following theorem, stating the soundness of the logic presented in Section 2.3.

Theorem 4.1 (Soundness) *Assume $I \vdash \{P\}\ S\ \{Q\}$, where either*

- *resource invariants in I are precise and the $*$ operation is cancellative; or*
- CONJ *is not used in the derivation of the triple.*

Then for any $\sigma_0 \in \Sigma$ such that

$$\sigma_0 \in P * \left(\circledast_{k=1}^{m} [\![I_k]\!] \right), \tag{4}$$

the program S is safe when run from σ_0, and if $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc_f}, \sigma$, we have

$$\sigma \in Q * \left(\circledast_{k=1}^{m} [\![I_k]\!] \right). \tag{5}$$

4.1 Thread-local interpretation and Separation Property

A *semantic proof* is defined as a triple $(C, G, \mathcal{I})$, where

- C is a command with a CFG $(N, T, \mathsf{start}, \mathsf{end})$;
- $G : N \to \mathcal{P}(\Sigma)$ maps program points of C to semantic annotations;
- $\mathcal{I} \in (\mathcal{P}(\Sigma))^m$ is a vector of resource invariant denotations $\mathcal{I}_k \in \mathcal{P}(\Sigma)$, $k = 1..m$

such that for all edges $(v, C', v') \in T$

- if $C' \in \mathsf{PComm}$, then
$$f_{C'}(G(v)) \sqsubseteq G(v'); \tag{6}$$
- if C' is $\texttt{acquire}(\ell_k)$, then
$$G(v) * \mathcal{I}_k \subseteq G(v'); \tag{7}$$
- if C' is $\texttt{release}(\ell_k)$, then
$$G(v) \subseteq G(v') * \mathcal{I}_k. \tag{8}$$

Note that the elements of $\mathcal{P}(\Sigma)$ assigned to program points by the semantic annotation mapping G in this definition are similar to label invariants in proof systems for unstructured control flow [7]. Inequalities (6), (7) and (8) are semantic counterparts of the axioms PRIM and the global versions of ACQUIRE and RELEASE, respectively.

The *thread-local interpretation* of a command is given by its semantic proof. In Section 4.2 we show how to extract a semantic proof for a thread in the program from its syntactic proof in concurrent separation logic.

As we explained in Section 1, the core of our proof of soundness consists of establishing the Separation Property [12]: at any time, the state of the program can be partitioned into that owned by each thread and each free lock. The following lemma formalises the property in the case where the local states of threads are defined by their semantic proofs. This establishes a correspondence between our thread-local interpretation and the operational semantics of Section 3.

Lemma 4.2 (Separation Property) *Assume semantic proofs* $(C_k, G_k, \mathcal{I})$, $k = 1..n$. *If* $\sigma_0 \in \Sigma$ *is such that*

$$\{\sigma_0\} \sqsubseteq \left(\circledast_{k=1}^{n} G_k(\mathsf{start}_k)\right) * \left(\circledast_{k\in\{1,\ldots,m\}} \mathcal{I}_k\right), \tag{9}$$

then, whenever $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, *we have*

$$\{\!|\sigma|\!\} \sqsubseteq \left(\circledast_{k=1}^{n} G_k(\mathsf{pc}(k))\right) * \left(\circledast_{k\in\mathsf{Free}(\mathsf{pc})} \mathcal{I}_k\right). \tag{10}$$

Proof We prove the statement of the theorem by induction on the length of the derivation of σ in the operational semantics of the program S. In the base case (10) is equivalent to (9). Suppose now that

$$\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}[j : v], \sigma \to_S \mathsf{pc}[j : v'], \sigma'.$$

Then $(v, C, v') \in T$ for some atomic command $C \in \mathsf{Comm}$. We have to show that if

$$\{\sigma\} \sqsubseteq \left(\circledast_{k=1}^{n} G_k((\mathsf{pc}[j : v])(k))\right) * \left(\circledast_{k\in\mathsf{Free}(\mathsf{pc}[j:v])} \mathcal{I}_k\right), \tag{11}$$

then

$$\{\!|\sigma'|\!\} \sqsubseteq \left(\circledast_{k=1}^{n} G_k((\mathsf{pc}[j : v'])(k))\right) * \left(\circledast_{k\in\mathsf{Free}(\mathsf{pc}[j:v'])} \mathcal{I}_k\right). \tag{12}$$

There are three cases corresponding to the type of the command C.

Case 1. $C \in \mathsf{PComm}$. In this case $\mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v'])$. Let

$$r = \left(\circledast_{\substack{1 \le k \le n, \\ k \neq j}} G_k(\mathsf{pc}(k)) \right) * \left(\circledast_{k \in W} \mathcal{I}_k \right), \tag{13}$$

where $W = \mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v'])$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &\sqsubseteq f_C(\{\sigma\}) && \text{definition of } \rightarrow_S \\ &\sqsubseteq f_C(G_j(v) * r) && (11) \\ &\sqsubseteq f_C(G_j(v)) * r && f_C \text{ is local} \\ &\sqsubseteq G_j(v') * r && (6) \end{aligned}$$

Case 2. C *is* $\mathtt{acquire}(\ell_i)$. In this case $i \in \mathsf{Free}(\mathsf{pc}[j : v])$ and $i \notin \mathsf{Free}(\mathsf{pc}[j : v'])$. Let r be defined by (13) with $W = \mathsf{Free}(\mathsf{pc}[j : v]) \setminus \{i\} = \mathsf{Free}(\mathsf{pc}[j : v'])$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &= \{\sigma\} && \text{definition of } \rightarrow_S \\ &\sqsubseteq G_j(v) * \mathcal{I}_i * r && (11) \\ &\sqsubseteq G_j(v') * r && (7) \end{aligned}$$

Case 3. C *is* $\mathtt{release}(\ell_i)$. In this case $i \notin \mathsf{Free}(\mathsf{pc}[j : v])$ and $i \in \mathsf{Free}(\mathsf{pc}[j : v'])$. Let r be defined by (13) with $W = \mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v']) \setminus \{i\}$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &= \{\sigma\} && \text{definition of } \rightarrow_S \\ &\sqsubseteq G_j(v) * r && (11) \\ &\sqsubseteq G_j(v') * \mathcal{I}_i * r && (8) \end{aligned}$$

In all cases we get inequalities equivalent to (12), completing the induction. □

4.2 Soundness with respect to thread-local interpretation

To prove Theorem 4.1, we first define a notion of validity of Hoare triples with respect to the thread-local interpretation of Section 4.1 and prove the soundness of the proof rules in this interpretation. Soundness of the logic with respect to the concrete semantics is then a direct consequence of Lemma 4.2.

Definition 4.3 *We write* $I \vDash \{P\}\ C\ \{Q\}$ *if there exists a semantic proof* $(C, G, [\![I]\!])$ *such that* $G(\mathsf{start}) = [\![P]\!]$ *and* $G(\mathsf{end}) \subseteq [\![Q]\!]$, *where* start *and* end *are the starting and the final program points of the CFG of* C.

We say that an inference rule is sound with respect to the thread-local interpretation if it preserves validity of judgements (as defined by the relation $\vDash$ above).

Lemma 4.4 PRIM, ACQUIRE, RELEASE, SEQ, CHOICE, LOOP *and* CONSEQ *are sound with respect to the thread-local interpretation.*

Proof For illustration, we consider the cases PRIM and SEQ of this easy lemma and omit the others.

PRIM. Consider an application $I \vdash \{P\}\ C\ \{Q\}$ of the axiom PRIM, where $C \in \mathsf{PComm}$. Then $f_C([\![P]\!]) \sqsubseteq [\![Q]\!]$. According to the encoding of commands into CFGs from Section 3, the command C has the CFG $(\{\mathsf{start}, \mathsf{end}\}, \{(\mathsf{start}, C, \mathsf{end})\}, \mathsf{start}, \mathsf{end})$. Let $G(\mathsf{start}) = [\![P]\!]$ and $G(\mathsf{end}) = [\![Q]\!]$, then $f_C(G(\mathsf{start})) \sqsubseteq G(\mathsf{end})$. Thus, $(G, C, [\![I]\!])$ is a semantic proof and $I \vDash \{P\}\ C\ \{Q\}$ as required.

SEQ. Assume $I \vDash \{P\}\ C_1\ \{Q\}$ and $I \vDash \{Q\}\ C_2\ \{R\}$. Let the CFGs of C_1 and C_2 be $(N_1, T_1, \mathsf{start}_1, \mathsf{end}_1)$ and $(N_2, T_2, \mathsf{start}_2, \mathsf{end}_2)$, respectively. Then $C_1; C_2$ has the CFG $(N_1 \cup N_2, T_1 \cup T_2 \cup \{(\mathsf{end}_1, \mathtt{skip}, \mathsf{start}_2)\}, \mathsf{start}_1, \mathsf{end}_2)$. There exist semantic proofs $(C_1, G_1, [\![I]\!])$ and $(C_2, G_2, [\![I]\!])$ such that

$$G_1(\mathsf{start}_1) = [\![P]\!], \quad G_1(\mathsf{end}_1) \subseteq [\![Q]\!], \quad G_2(\mathsf{start}_2) = [\![Q]\!], \quad G_2(\mathsf{end}_2) \subseteq [\![R]\!].$$

Let $G(v) = G_1(v)$ for $v \in N_1$ and $G(v) = G_2(v)$ for $v \in N_2$. We have $f_{\mathtt{skip}}(G_1(\mathsf{end}_2)) \sqsubseteq G_2(\mathsf{start}_1)$. Thus, $(G, (C_1; C_2), [\![I]\!])$ is a semantic proof and $I \vDash \{P\}\ C_1; C_2\ \{R\}$. □

We now proceed to prove the soundness of the rules FRAME, DISJ and CONJ. To this end, we show that we can construct semantic proofs for the conclusions of these rules from semantic proofs for their premisses. This is essentially a semantic counterpart of a proof that these rules are admissible in the logic including the global ACQUIRE and RELEASE axioms, i.e., that a derivation using these rules can be converted into a derivation that does not use them. By using semantic proofs instead of derivations in our proof system, we avoid having to deal with the syntactic form of the proof rules in the logic and the control-flow constructs in our programming language.

Lemma 4.5 *(i) For any $r \in \mathcal{P}(\Sigma)$, if $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G(v) * r$.*

(ii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \cup G_2(v)$.

(iii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \cap G_2(v)$, provided the resource invariant denotations in $\mathcal{I}$ are precise and the $*$ operation is cancellative.

Proof Consider an edge (v, C', v') in the CFG of the command C. When $C' \in \mathsf{PComm}$, inequality (6) for the new semantic annotation G' follows from the fact that the predicate transformer $f_{C'}$ is local and distributes over $\sqcup$ and $\sqcap$. The latter is true by construction of transformers defined by pointwise lifting from Σ; see (1) and (2). We omit the easy case when C' is $\mathtt{acquire}(\ell_k)$. Suppose now that C' is $\mathtt{release}(\ell_k)$. We consider every case of the lemma in turn.

(i) By the definition of G', we have

$$G'(v) = G(v) * r \subseteq G(v') * \mathcal{I}_k * r = G'(v') * \mathcal{I}_k.$$

(ii) The $*$ operation distributes over $\cup$:

$$\forall p_1, p_2, q \in \mathcal{P}(\Sigma).\ (p_1 \cup p_2) * q = (p_1 * q) \cup (p_2 * q).$$

Hence,

$$G'(v) = G_1(v) \cup G_2(v) \subseteq (G_1(v') * \mathcal{I}_k) \cup (G_2(v') * \mathcal{I}_k) = (G_1(v') \cup G_2(v')) * \mathcal{I}_k = G'(v') * \mathcal{I}_k.$$

(iii) It is well-known [4] that if $*$ is cancellative, then for a precise $q \in \mathcal{P}(\Sigma)$ and any $p_1, p_2 \in \mathcal{P}(\Sigma)$ we have

$$(p_1 \cap p_2) * q = (p_1 * q) \cap (p_2 * q). \tag{14}$$

Thus, in this case we establish

$$G'(v) = G_1(v) \cap G_2(v) \subseteq (G_1(v') * \mathcal{I}_k) \cap (G_2(v') * \mathcal{I}_k) = (G_1(v') \cap G_2(v')) * \mathcal{I}_k = G'(v') * \mathcal{I}_k.$$

In all cases we get (8), which completes the proof. □

Corollary 4.6 *The rules* Frame *and* Disj *are sound with respect to the thread-local interpretation. So is* Conj *when the resource invariants in I are precise and the $*$ operation is cancellative.*

Lemma 4.7 *If $I \vdash \{P\}\ C\ \{Q\}$ and the restrictions on the derivation from Theorem 4.1 hold, then $I \vDash \{P\}\ C\ \{Q\}$.*

The proof is by induction on the derivation of $I \vdash \{P\}\ C\ \{Q\}$ using Lemma 4.4 and Corollary 4.6.

4.3 The proof

Proof of Theorem 4.1 Let $I \vdash \{P_k\}\ C_k\ \{Q_k\}$ be the thread-local triples used in the rule Par to derive $I \vdash \{P\}\ S\ \{Q\}$. Then $[\![P]\!] = (\circledast_{k=1}^{n} [\![P_k]\!])$ and $[\![Q]\!] = (\circledast_{k=1}^{n} [\![Q_k]\!])$. By Lemma 4.7, $I \vDash \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$, hence, by Definition 4.3, there exist semantic proofs $(C_k, G_k, [\![I]\!])$, $k = 1..n$ such that $G_k(\mathsf{start}_k) = [\![P_k]\!]$ and $G_k(\mathsf{end}_k) \subseteq [\![Q_k]\!]$. Consider a state σ_0 satisfying (4). Let $\mathcal{I} = [\![I]\!]$ in Lemma 4.2, then (9) is fulfilled. We have $\forall v.\, G_k(v) \neq \top$. Therefore, for any pc and σ such that $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$, from (10) we get $\{\!|\sigma|\!\} \sqsubset \top$, i.e., S is safe when run from σ_0. Now letting $\mathsf{pc} = \mathsf{pc}_\mathbf{f}$ in (10), we get (5). □

Note that (14) does not hold in general for imprecise q. Thus, the conjunction of two semantic proofs is not necessarily a semantic proof, and Lemma 4.5 may not

be extended to show the soundness of the conjunction rule in the case of imprecise resource invariants. This is expected: the famous Reynolds counterexample [12] shows that in this case concurrent separation logic is unsound.

The intuitive reason for the unsoundness is that imprecise resource invariants allow splitting the heap at a `release` command in different ways in different branches of the proof. Hence, in the two premisses of Conj there may be different understandings of what the partitioning of the global heap into thread-local and protected parts should be. Trying to $\wedge$-conjoin two such judgements about the local state of a thread then leads to inconsistency. Note that the simplicity of the interpretation using semantic proofs comes from not abstracting away from the crucial partitioning choice details. Thus, conflicting partitioning choices lead to directly conflicting semantic proofs.

4.4 Logical variables

Assume that Σ is an algebra with logical variables, i.e., $\Sigma = \Sigma' \times \text{Ints}$, and that the functions f_C for $C \in \mathsf{PComm}$ are lifted from functions on Σ' (Section 2.4). In this case, we can add to the logic the rules Exists and Forall. We say that $p \in \mathcal{P}(\Sigma)$ *does not depend on the interpretation of logical variables*, if for any $(\sigma', \mathbf{i}) \in p$ and $\mathbf{i}' \in \text{Ints}$ we have $(\sigma', \mathbf{i}') \in p$. For the rules to be sound in a concurrent setting, we must require that the denotations of resource invariants do not depend on interpretations. In the case when we include Forall, we must additionally require that the resource invariants be precise and the $*$ operation be cancellative.

For a logical variable X let $\mathsf{Exists}(X) : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, respectively, $\mathsf{Forall}(X) : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ be the semantic counterparts of existential, respectively, universal quantification of X, defined as follows:

$$\mathsf{Exists}(X, p) = \{(\sigma', \mathbf{i}) \mid \exists u.\, (\sigma', \mathbf{i}[X : u]) \in p\};$$
$$\mathsf{Forall}(X, p) = \{(\sigma', \mathbf{i}) \mid \forall u.\, (\sigma', \mathbf{i}[X : u]) \in p\}.$$

The proof of soundness of Exists and Forall with respect to the thread-local interpretation is done by establishing the following analogue of Lemma 4.5.

Lemma 4.8 *Under the above conditions, for any logical variable X and resource invariants $\mathcal{I}$ that do not depend on interpretations:*

(i) If $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = \mathsf{Exists}(X, G(v))$.

(ii) If $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = \mathsf{Forall}(X, G(v))$, provided the resource invariant denotations in $\mathcal{I}$ are precise and the $*$ operation is cancellative.

The proof is similar to that of Lemma 4.5. It follows that Theorem 4.1 holds for the logic extended with the rules Exists and Forall, subject to the conditions given above.

5 Data-race freedom

We now show that the provability of a program in our logic implies that the program has no data races. We formalise the notion of data-race freedom and prove this result for the case when $\Sigma = \mathsf{RAM}$ (Section 2.1) and $\mathsf{PComm} = \mathsf{RAMComm}$ (Section 2.2).

For a state $\sigma \in \Sigma$ let $\mathsf{accesses}(C, \sigma)$, respectively, $\mathsf{writes}(C, \sigma)$ be the set of variables and locations that a primitive sequential command $C \in \mathsf{RAMComm}$ may access (i.e., read, write, or dispose), respectively, write to or dispose, when run from the state σ according to the semantics of commands RAMComm defined in Figure 1.

Definition 5.1 (Interfering commands) *Commands C' and C'' from* RAMComm *interfere with each other when executed from the state $\sigma \in \mathsf{RAM}$, denoted with $C' \bowtie_\sigma C''$, if*

$$(\mathsf{accesses}(C', \sigma) \cap \mathsf{writes}(C'', \sigma) \neq \emptyset) \ \vee \ (\mathsf{writes}(C', \sigma) \cap \mathsf{accesses}(C'', \sigma) \neq \emptyset).$$

Given this formulation of interference, the usual notion of data races is formulated as follows.

Definition 5.2 (Data race) *Program S has a data race when run from an initial state $\sigma_0 \in \mathsf{RAM}$ if for some i, j and pc such that $i \neq j$, $\mathsf{pc}(i) = v_i$ and $\mathsf{pc}(j) = v_j$ and state $\sigma \in \mathsf{RAM}$ such that $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$, there exist CFG edges $(v_i, C', v_i') \in T_i$ and $(v_j, C'', v_j') \in T_j$ in the control-flow relations of threads i and j labelled with commands C' and C'' from* RAMComm *such that*

$$C', \sigma \not\rightarrow \top; \quad C'', \sigma \not\rightarrow \top; \quad C' \bowtie_\sigma C''. \tag{15}$$

We first prove that the existence of a thread-local interpretation for a program (as defined in Lemma 4.2) implies data-race freedom.

Lemma 5.3 *Under the conditions of Lemma 4.2 with $\Sigma = \mathsf{RAM}$ and $\mathsf{PComm} = \mathsf{RAMComm}$, the program S has no data races when run from initial states σ_0 satisfying (9).*

Proof Suppose the contrary: there exist i, j and pc such that $i \neq j$, $\mathsf{pc}(i) = v_i$ and $\mathsf{pc}(j) = v_j$, a state σ such that $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$, CFG edges $(v_i, C', v_i') \in T_i$ and $(v_j, C'', v_j') \in T_j$ labelled with commands C' and C'' from RAMComm such that (15) holds.

By Lemma 4.2, $\sigma \in r * G(v_i) * G(v_j)$ for some r. Hence,

$$\sigma = \sigma_0 * \sigma_1 * \sigma_2, \tag{16}$$

where

$$\sigma_0 \in r, \quad \sigma_1 \in G(v_i), \quad \sigma_2 \in G(v_j). \tag{17}$$

Since the values of G are distinct from $\top$, it follows that $f_{C'}(G(v_i)) \sqsubset \top$ and $f_{C''}(G(v_j)) \sqsubset \top$. From this and (17) we obtain $f_{C'}(\sigma_1) \sqsubseteq f_{C'}(G(v_i)) \sqsubset \top$. So, $f_{C'}(\sigma_1) \sqsubset \top$ and, analogously, $f_{C''}(\sigma_2) \sqsubset \top$. Hence, $C', \sigma_1 \not\rightarrow \top$ and $C'', \sigma_2 \not\rightarrow \top$.

From this and the fact that $C' \bowtie_\sigma C''$ using the definitions of $*$ and the predicate transformers for commands in RAMComm, we easily get that $\sigma_1 * \sigma_2$ is undefined, which contradicts (16). The intuition behind this is that from $C', \sigma_1 \not\rightarrow \top$ and $C'', \sigma_2 \not\rightarrow \top$ it follows that both σ_1 and σ_2 should be defined on the same variable or location accessed by C' and C'', which makes the state $\sigma_1 * \sigma_2$ inconsistent. □

As a corollary of Lemma 5.3, we easily get the data-race freedom theorem.

Corollary 5.4 (Data-race freedom) *Under the conditions of Theorem 4.1 with* $\Sigma = \mathsf{RAM}$ *and* $\mathsf{PComm} = \mathsf{RAMComm}$*, the program* S *has no data races when run from initial states* σ_0 *satisfying (4).*

6 Conclusion

Conceptually, the idea of our proof of soundness is simple: we show the Separation Property by induction on a derivation in the operational semantics. Our proof expresses this property directly as an invariant preserved by concurrent executions, and thus does not need to unpick concurrent executions into constituent sequential ones and there track changing splittings.

Our use of semantic proofs is inspired by program analyses based on abstract interpretation [5], which compute mappings from program points to elements of an abstract domain denoting sets of states. Proofs of soundness for such analyses rely crucially on the intentional information provided by the mappings. In fact, our proof of the Separation Property (Lemma 4.2) is almost identical to the proof soundness of a program analysis for inferring resource invariants in concurrent separation logic we have previously developed [9]. The aim of this paper has been to argue that the approach based on semantic proofs is also useful in proving the soundness of program logics and to demonstrate the corresponding techniques in a clean setting.

In this paper, we presented our results for a simplistic programming language. However, we have also applied our approach to more expressive languages, including dynamic lock allocation and deallocation, dynamic thread creation and first-order procedures; see [8]. Additionally, we have applied it to prove the soundness of a logic for verifying preemptive OS kernels [10], which establishes a form of refinement. From our experience, the approach provides a low-cost way of proving the soundness of complicated concurrency logics.

An interesting question for future research raised by our results is whether there are other ways to ensure the soundness of the conjunction rule other than requiring precision. After all, equation (14), which is the only place in the proof where precision is used, may be satisfied even if the predicate q in it is imprecise. Intuitively, for the conjunction rule to be sound, the proofs being combined have to split the state in the same way at every `release` command in the program. One way to enforce this is to require that the postcondition of `release` in the proofs be computed as a function of the precondition. Unfortunately, our preliminary investigations show that straightforward formulations of such functions may invalidate the frame rule. The possible fixes are not pretty, and the appropriate solution seems to depend on

the particular class of programs considered.

Acknowledgement

We would like to thank Peter O'Hearn, Hongseok Yang and the anonymous reviewers for helpful comments and suggestions.

References

[1] Bornat, R., C. Calcagno, P. W. O'Hearn and M. Parkinson, *Permission accounting in separation logic*, in: *POPL'05: Symposium on Principles of Programming Languages* (2005), pp. 259–270.

[2] Brookes, S. D., *A semantics of concurrent separation logic*, Theor. Comput. Sci. **375** (2007), pp. 227–270, preliminary version appeared in *CONCUR'04: Conference on Concurrency Theory*.

[3] Calcagno, C., D. Distefano and V. Vafeiadis, *Compositional resource invariant synthesis*, in: *APLAS'09: Asian Symposium on Programming Languages and Systems* (2009).

[4] Calcagno, C., P. W. O'Hearn and H. Yang, *Local action and abstract separation logic*, in: *LICS'07: Symposium on Logic in Computer Science* (2007), pp. 366–378.

[5] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL'77: Symposium on Principles of Programming Languages* (1977), pp. 238–252.

[6] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *POPL'79: Symposium on Principles of Programming Languages* (1979), pp. 269–282.

[7] de Bruin, A., *Goto statements: Semantics and deduction systems*, Acta Inf. **15** (1981), pp. 385–424.

[8] Gotsman, A., *Logics and analyses for concurrent heap-manipulating programs* (2009), phD Thesis. Technical Report UCAM-CL-TR-758, University of Cambridge Computer Laboratory.

[9] Gotsman, A., J. Berdine, B. Cook and M. Sagiv, *Thread-modular shape analysis*, in: *PLDI'07: Conference on Programming Languages Design and Implementation* (2007), pp. 266–277.

[10] Gotsman, A. and H. Yang, *Modular verification of preemptive OS kernels*, in: *IFCP'11: International Conference on Functional Programming*, 2011, to appear.

[11] Hayman, J. and G. Winskel, *Independence and concurrent separation logic*, in: *LICS'06: Symposium on Logic in Computer Science* (2006), pp. 147–156.

[12] O'Hearn, P. W., *Resources, concurrency and local reasoning*, Theor. Comput. Sci. **375** (2007), pp. 271–307, preliminary version appeared in *CONCUR'04: Conference on Concurrency Theory*.

[13] O'Hearn, P. W., H. Yang and J. C. Reynolds, *Separation and information hiding*, ACM Trans. Program. Lang. Syst. **31** (2009), pp. 1–50, preliminary version appeared in *POPL'04: Symposium on Principles of Programming Languages*.

[14] Parkinson, M., R. Bornat and C. Calcagno, *Variables as resource in Hoare logics*, in: *LICS'06: Symposium on Logic in Computer Science* (2006), pp. 137–146.

[15] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *LICS'02: Symposium on Logic in Computer Science* (2002), pp. 55–74.

[16] Yang, H., O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano and P. O'Hearn, *Scalable shape analysis for systems code*, in: *CAV'08: Conference on Computer Aided Verification*, LNCS **5123** (2008).

[17] Yang, H. and P. W. O'Hearn, *A semantic basis for local reasoning*, in: *FOSSACS'02: Conference on Foundations of Software Science and Computation Structures*, LNCS **2303** (2002), pp. 402–416.